Methodologies for Specification of Real-Time Systems Using Timed I/O Automata

David, Alexandre; Larsen, Kim Guldstrand; Legay, Axel; Nyman, Ulrik; Wasowski, Andrzej

Published in:
Lecture Notes in Computer Science

DOI (link to publication from Publisher):
10.1007/978-3-642-17071-3_15

Publication date:
2010

Document Version
Other version

Methodologies for Specification of Real-Time Systems Using Timed I/O Automata

Alexandre David[1], Kim G. Larsen[1], Axel Legay[2],
Ulrik Nyman[1], Andrzej Wąsowski[3]

[1] Computer Science, Aalborg University, Denmark
[2] INRIA/IRISA, Rennes Cedex, France
[3] IT University of Copenhagen, Denmark

Abstract. We present a real-time specification framework based on Timed I/O Automata and a comprehensive tool support for it. The framework supports various design methodologies including: top-down refinement—for decomposition of abstract specifications towards increasingly detailed models; bottom-up abstraction—for synthesis of complex systems from more concrete models; and step-wise modularisation of requirements—to factor out behaviours given by existing available components from a complex global requirements specification to be implemented. These methodologies are realized by consecutive applications of operators from the following set: refinement, consistency checking, logical and structural composition and quotienting. Additionally, our tool allows combining the component-oriented design process with verification of temporal logic properties increasing the flexibility of the process.

1 Context and Motivation

Industries developing complex embedded systems, such as aerospace and automotive, have undergone deep organisational changes with tremendous impact on development processes. In the past, they were vertically integrated companies, internally supporting all design activities from specification to implementation. Today they rely increasingly on external suppliers and on independent teams to provide essential components of systems. It is no longer possible for a single team to control the entire design process from specification to implementation.

Complex systems emerge from assembling multiple components. These components are designed by independently working teams, who adhere to a common agreement, a *contract*, on what the interface of each component should be. Such an interface defines the behaviours expected from the component as well as the environment in which it can be used. The main advantage is that it abstracts from the way the component can be implemented.

In practice interfaces are described using textual documents or modelling languages such as UML or WSDL. Unfortunately, such specifications are ambiguous and thus are subject to interpretation. We instead recommend relying on mathematically sound formalisms to reduce ambiguities. In this context, the vibrant research area of *compositional reasoning* [20] gives the foundations that allow to

reason about properties of the global system based on properties of individual components. The essential advantage of compositional reasoning is its support for safe reuse of components, well known from other engineering disciplines.

Building specification theories is a subject of intensive studies [11, 14]. One particularly successful direction are interface automata [14, 15, 23, 31]. In this framework, an interface is represented by an input/output automaton [29], where transitions are typed as *input* and *output*. The semantics is given by a two-player game: the *input* player represents the environment, and the *output* player represents the component itself. Contrary to the input/output model of [29], this semantic offers an optimistic treatment of composition: two interfaces can be composed if there exists an environment in which they can safely interact.

The existing interface theories focus primarily on composition (and sometimes on refinement). There hardly exist supporting tools that could be used by engineers. Over the years of interaction with industrial partners, we have collected the following requirements for interfaces theories. Notice that they significantly exceed the usual scope of studying composition and refinement.

1. It should be decidable whether an interface admits an implementation.[4]
2. There must be a mechanism to safely replace a component by another one. Technically this corresponds to the requirements of precongruence and completeness for *Refinement*. Refinement (written $\leq$), which is a preorder on the set of interfaces, should satisfy the following property:
   Every implementation satisfying a refinement of an interface should also satisfy this interface.
3. To control design complexity, one should be able to decide whether there exists an interface that refines two different interfaces (a *shared refinement*).
4. Different aspects of systems are often specified by different teams. The issue of dealing with multiple aspects or multiple viewpoints is thus essential. It should be possible to represent several interfaces (viewpoints) for the same component, and to combine then in a conjunctive fashion. Conjunction (written $\wedge$) should satisfy the following property:
   Given two viewpoints represented by two interfaces, any implementation that satisfies the conjunction must satisfy the two viewpoints.
5. The framework should provide a combination operation reflecting the standard interaction between systems. It should respect the refinement to support independent development:
   Given two implementations of two interfaces, the composition of the implementations satisfies the composition of their interfaces.
6. It should be possible to factor in existing components into general requirements, in order to facilitate reuse of accumulated assets. In interface theories this is realized using a quotient operator.
7. Conjunction and composition must be associative and commutative, so that the emergent behaviour of the system depends only on the specifications, not on the order in which they have been combined.

[4] In our theory, an implementation shall not be viewed as a program in a concrete programming language but rather as an abstract mathematical object that represents a set of programs sharing common properties.

8. There must exist a specification language to specify properties of interfaces as well as a procedure to decide whether the interface satisfies the properties.
9. All the above operations and properties should be performed and checked with efficient algorithms.
10. User-friendly tools providing comprehensible feedback to the user must be available. For example, if an implementation violates a specification, a useful feedback inspires the designer on how to correct it.

In [16], a timed extension of the theory of interface automata has been introduced, motivated by the fact that time can be a crucial parameter in practice, for example in embedded systems. The results of [16] focus mostly on structural composition. Recently [12] we have proposed what seems to be the first complete interface theory for timed systems (with respect to the above requirements). Our specifications are *timed* input/output automata [21]—timed automata whose sets of discrete transitions are split into *input* and *output* transitions. Contrary to [16] and [21] our theory distinguishes between implementations and specifications. This is done by assuming that the former have fixed timing behaviour and they can always advance either by producing an output or delaying. The theory also provides a game-based algorithm to decide whether a specification is consistent, i.e. whether it has at least one implementation. The latter reduces to deciding existence of a strategy that despite the behaviour of the environment will avoid states that cannot possibly satisfy the implementation requirements.

A *pruning* facility removes all the states not covered by the strategy. It can drastically reduce the state-space of the system. Following a similar principle, it is possible to constrain an interface with a timed temporal logic formula [1]. For example, like in [16], one can use a Büchi objective to remove states allowing Zeno behaviours. Our theory is rich in the sense that it captures all the good operations for a compositional design theory presented above. Also all the algorithms have been implemented. This implementation (available at [36]) comes as an extension of the Uppaal-tiga tool-set [3]. Uppaal-tiga is a tool that implements a series of algorithms for solving timed games [9] as well as checking timed temporal logic properties. Working within Uppaal-tiga allows us to propose a state-of-the-art user interface for verification tools.

In this paper our objectives are (1) to give more insight into design choices made in [12], (2) to report on challenges of the implementation, (3) to discuss design methodologies compatible with our theory, (4) to evaluate the implementation, and (5) to compare our results with other results in the same field.

2 Specifications and Implementations

We shall now introduce our component model.

Definition 1. *A Timed I/O Transition System (TIOTS) is a quadruple* $S = (St^S, s_0, \Sigma^S, \rightarrow^S)$*, where* St^S *is an infinite set of states,* $s_0 \in St$ *is the initial state,* $\Sigma^S = \Sigma^S_{\mathrm{i}} \oplus \Sigma^S_{\mathrm{o}}$ *is a finite set of actions partitioned into inputs* (Σ^S_{i}) *and outputs* (Σ^S_{o}) *and* $\rightarrow^S \subseteq St^S \times (\Sigma^S \cup \mathbb{R}_{\geq 0}) \times St^S$ *is a transition relation. We*

write $s \xrightarrow{a}^S s'$ *instead of* $(s,a,s') \in \rightarrow^S$ *and use* $i?$, $o!$ *and* d *to range over inputs, outputs and* $\mathbb{R}_{\geq 0}$ *respectively. In addition any TIOTS satisfies the following:*

[time determinism] whenever $s \xrightarrow{d}^S s'$ *and* $s \xrightarrow{d}^S s''$ *then* $s' = s''$

[time reflexivity] $s \xrightarrow{0}^S s$ *for all* $s \in St^S$

[time additivity] for all $s, s'' \in St^S$ *and all* $d_1, d_2 \in \mathbb{R}_{\geq 0}$ *we have* $s \xrightarrow{d_1+d_2}^S s''$ *iff* $s \xrightarrow{d_1}^S s'$ *and* $s' \xrightarrow{d_2}^S s''$ *for an* $s' \in St^S$

TIOTSs are semantic objects that represent timed interactive processes. In our framework we use *Timed I/O Automata* as a syntactic domain in which *specifications* and *implementations* are represented.

Definition 2. *A* Timed I/O Automaton *(TIOA) is a tuple* $A = (Loc, q_0, Clk, E, Act, Inv)$ *where Loc is a finite set of locations,* $q_0 \in Loc$ *is the initial location, Clk is a finite set of clocks,* $E \subseteq Loc \times Act \times \mathcal{B}(Clk) \times \mathscr{P}(Clk) \times Loc$ *is a set of edges with* $\mathcal{B}(Clk)$ *being a set of clock constraints,* $Act = Act_{\mathrm{i}} \oplus Act_{\mathrm{o}}$ *is a finite set of actions, partitioned into inputs and outputs respectively, and* $Inv : Loc \mapsto \mathcal{B}(Clk)$ *is a set of location invariants.*

As for timed automata, a *state* of A is a pair (q, V) where q is a location and $V : Clk \mapsto \mathbb{R}_{\geq 0}$ is a *valuation function* that assigns a non-negative value to each clock in *Clk*. We write $u + d$ to denote a valuation such that for any clock r we have $(u + d)(r) = x + d$ iff $u(r) = x$. Given $d \in \mathbb{R}_{\geq 0}$, we write $u[r \mapsto 0]_{r \in c}$ for a valuation which agrees with u on all values for clocks not in c, and returns 0 for all clocks in c. We use $\mathbf{0}$ to denote the constant function mapping all clocks to zero. The *initial state* of A is the pair $(q_0, \mathbf{0})$.

We visualise TIOAs using classical Timed Automata notation, extending it with two types of transitions (inputs and outputs). See example in Figure 1.

The semantics of a TIOA $A = (Loc, q_0, Clk, E, Act, Inv)$ is a TIOTS $[\![A]\!]_{\mathrm{sem}} = (Loc \times (Clk \mapsto \mathbb{R}_{\geq 0}), (q_0, \mathbf{0}), Act, \rightarrow)$, where $\rightarrow$ is the largest transition relation generated by the following rules:

- Each $(q, a, \varphi, c, q') \in E$ gives rise to $(q, u) \xrightarrow{a} (q', u')$ for each clock valuation $u \in [Clk \mapsto \mathbb{R}_{\geq 0}]$ such that $u \models \varphi$ and $u' = u[r \mapsto 0]_{r \in c}$ and $u' \models Inv(q')$.
- Each location $q \in Loc$ with a valuation $u \in [Clk \mapsto \mathbb{R}_{\geq 0}]$ gives rise to a transition $(q, u) \xrightarrow{d} (q, u + d)$ for each delay $d \in \mathbb{R}_{\geq 0}$ such that $u + d \models Inv(q)$.

Observe that the TIOTSs induced by a TIOAs naturally satisfy the three axioms of Definition 1. In the rest of the paper, we will only consider deterministic TIOAs, whose corresponding TIOTSs are deterministic.

A TIOTS represents a two-player *timed game* [9]. The *Input* player (the environment) controls the input transitions of the TIOTS. The *Output* player (the system) controls the output transitions. The formal definitions of strategy and move outcomes for such a game are given in [12]. The set of *winning states* from which one of the players has a strategy to satisfy a safety or a reachability objective can be computed with algorithms presented in [9]—efficient symbolic versions of well-known controller synthesis algorithms of [30].

We now define implementations and specifications in terms of TIOAs.

Definition 3. *A specification automaton is a TIOA that is input-enabled, i.e., in each state all the inputs should be available.*

The assumption of input-enabledness, also seen in many interface theories [28, 18, 34, 37, 32], reflects our belief that an input cannot be prevented from being sent to a system, but it might be unpredictable how the system behaves after receiving it. Input-enabledness encourages explicit modelling of this unpredictability, and compositional reasoning about it; for example, deciding if an unpredictable behaviour of one component induces unpredictability of the entire system. Observe that it is easy to check whether a TIOA is input-enabled. In practice tools can interpret absent input transitions in at least two reasonable ways. First, they can be interpreted as ignored inputs, corresponding to location loops in the automaton. Second, they may be seen as unavailable ('blocking') inputs, which can be achieved by assuming implicit transitions to a designated error state. Later, in Section 4.2 we will call such a state *strictly undesirable* and give a rationale for this name.

The role of specifications in a specification theory is to abstract, or underspecify, sets of possible implementations. *Implementations* are concrete executable realizations of systems. We will assume that implementations of timed systems have fixed timing behaviour (outputs occur at predictable times) and systems can always advance either by producing an output or delaying. Formally:

Definition 4. *An implementation is a specification that satisfies the two following conditions:*

1. **Independent progress:** *implementations cannot get stuck in a state where it is up to the environment to induce progress. In each implementation state either an output is possible or one can delay until an output is enabled.*
2. **Output urgency:** *if an output is available, then it cannot be delayed.*

Since specifications and implementations are TIOAs, their semantics are still given in terms of TIOTs. We refer the interested reader to [12] for more details.

Example. Figure 1a specifies a vending machine that can serve tea or coffee. A possible implementation of this machine can be found in Figure 1b. Both automata are deterministic. Note that the output transitions of the implementation Impl arrive at a fixed moment in time and cannot be delayed, which guarantees output urgency (the invariant guarantees progress and the guard constrains the transition). Each time the output tea! from Idle to Idle is taken, the clock y is reset. Without this reset, independent progress would not be guaranteed for valuations of the clock y that are greater than 6.

We now introduce *refinement*—a notion of comparison between two specifications and a way to relate implementations to specifications. Refinement should satisfy the following *substitutability* condition. If A_S refines A_T, it should be possible to replace A_T with A_S in every context and obtain an equivalent system. Contrary to the other operations, refinement is defined at the level of TIOTSs.

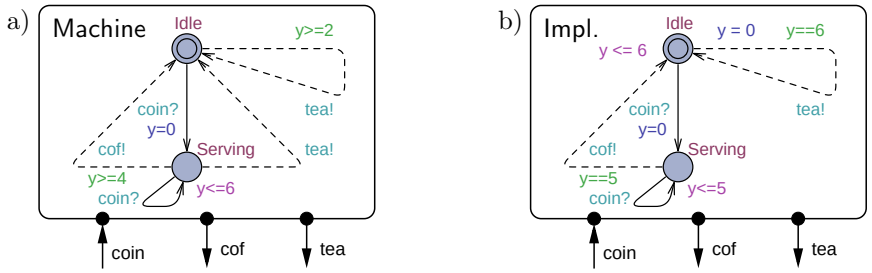

Fig. 1. a) Specification of a coffee and tea Machine and b) an implementation

Definition 5. *Let A_S and A_T be two specification automata and $S = (St^S, s_0, \Sigma, \rightarrow^S)$ and $T = (St^T, t_0, \Sigma, \rightarrow^T)$ be their corresponding timed transition systems. We say that A_S refines A_T, written $A_S \leq A_T$, iff there exists a binary relation $R \subseteq St^S \times St^T$ containing (s_0, t_0) and for all states sRt implies:*

1. *whenever $t \xrightarrow{i?}{}^T t'$ for some $t' \in St^T$ then $s \xrightarrow{i?}{}^S s'$ and $s'Rt'$ for some $s' \in St^S$*
2. *whenever $s \xrightarrow{o!}{}^S s'$ for some $s' \in St^S$ then $t \xrightarrow{o!}{}^T t'$ and $s'Rt'$ for some $t' \in St^T$*
3. *whenever $s \xrightarrow{d}{}^S s'$ for $d \in \mathbb{R}_{\geq 0}$ then $t \xrightarrow{d}{}^T t'$ and $s'Rt'$ for some $t' \in St^T$*

It is easy to see that the refinement is reflexive and transitive, so it is a pre-order on the set of all specifications. Refinement can be checked for specification automata by reducing the problem to a specific refinement game, and using a symbolic representation to reason about it. See Section 5 for more details.

Satisfaction is a simple application of refinement. More precisely, we say that an implementation satisfies a specification automaton iff it refines this specification. As an example, observe that the automaton in Figure 1b is a refinement of the one in Figure 1a, and thus it is also an implementation of it.

The set of all implementations of A is denoted $[\![A]\!]_{\text{mod}}$. In [12], we have shown that the refinement relation is complete for our specification model, i.e, A_S refines A_T iff the set of implementations that satisfy A_S is included in the set of implementations that satisfy A_T.

A specification may be *locally inconsistent* in the sense that it may contain *bad states*, i.e., states that do not satisfy the *independent progress* property[5]. We say that a specification is *consistent*, and thus useful, if it admits at least one implementation. It is important to have a procedure to decide whether a specification admits at least one implementation. In [12], we have shown that this question reduces to the one of deciding if there exists a strategy for the system (Output player) to avoid reaching *bad states* in the specification. A *pruning* facility removes from the TIOA all the behaviours that are not covered by the strategy. It can drastically reduce the state-space of the automaton. In the rest of the paper, we assume that bad states are always pruned away.

[5] In section 4, we shall observe that the combination of two specifications without bad states may lead to a specification with bad states.

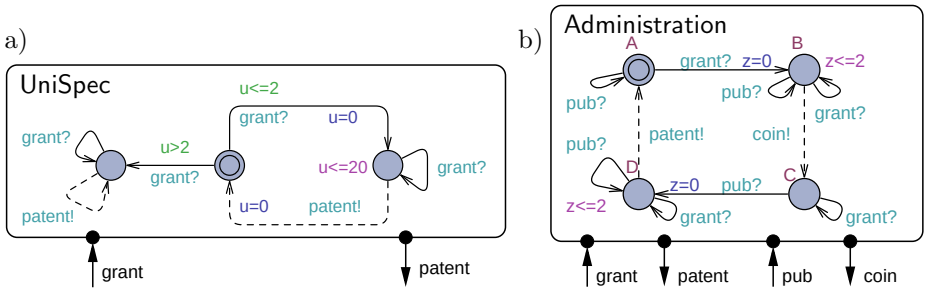

Fig. 2. a) University specification UniSpec. b) Specification of an Administration.

3 Design Methodologies

In the following we introduce three different development methodologies supported by our framework. These development methodologies are in no way in conflict with each other, but should more be seen as prototype work-flows that in a concrete development process would be combined. First we present the running example that will be used in presenting the methodologies.

The example is based on a very simplified view of a modern university. The purpose of the university is to file as many patents as possible. More precisely the requirements imposed on the university is given by the TIOA UniSpec as presented in Figure 2a. The border around the specification shows the input and output sort by incoming and outgoing arrows respectively. The initial state of the specification is marked by a double circled state. Given that the university receives a grant (solid transition marked with grant?) after a delay of less than two time units it will output (dashed transition marked with patent!) a patent within the next 20 time units. If the first grant comes after more than two time units or any subsequent grant comes more than two time units after a patent has been filed then the behaviour of the university becomes unpredictable, which is modelled by the leftmost state in the specification.

Stepwise Refinement. The first methodology presented is the classic top-down development through *stepwise decomposition and refinement*. Starting from the overall specification of Figure 2a one can refine this into a specification that contains several parallel components. The refinement is based on a knowledge of how the system under design is supposed to meet the overall requirements. This refined specification can again be refined further, until the desired level of detail has been reached. It is important to note that the *independent implementability* property allows for these refinement steps to be taken for individual components, greatly increasing the scalability of the framework through compositional design.

We will decompose the University specification into three components: an Administration, a Coffee/Tea machine and a Researcher. The responsibility of the Administration (Figure 2b) is to convert the grants provided to the University into coins that can be used in the coffee and tea machine. The coffee and tea

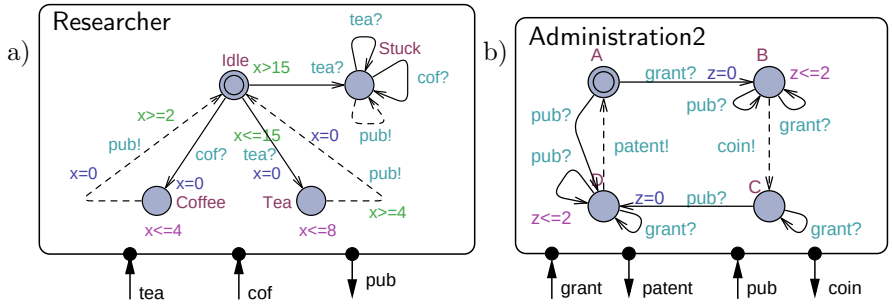

Fig. 3. Specifications for the a) Researcher and b) Administration2.

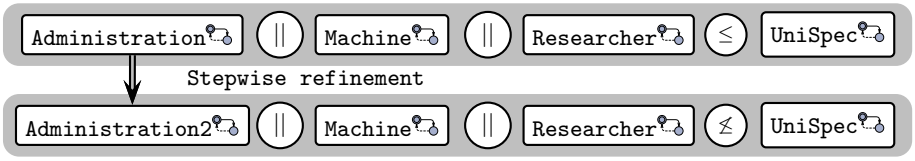

Fig. 4. One possible step in Top-down development: Modifying a component and rechecking refinement.

Machine (Figure 1a) will then in turn provide the Researcher (Figure 3a) with coffee and tea so that the researcher can produce publications. The Administration component also has the responsibility of converting publications into patents.

The top line in Figure 4 shows a successful refinement check which shows that the three components together refine the overall specification.

The bottom part of Figure 4 shows an additional step in the refinement process. Here a single component is updated (Administration to Administration2). This new version (Figure 3b) differs in a single transition. If it receives a publication (pub?) in the initial, left most state, then it will not loop but instead shift to the lower left state indicating that it is ready to output a patent based on this publication. This new version of the administration is thus able to receive and process free publications that it has not payed for.

Figure 4 shows that the refinement check fails after this update of the model. By the independent implementability property this could also have been discovered by checking whether Administration2 refines Administration, which indeed it does not. This might come as a surprise to the developer as it seems like a reasonable improvement to be able to accept free publications. We defer the discussion of how to solve this issue to Section 5.

In stepwise refinement this step of decomposing and refining individual components is applied iteratively, until a suitable level of detail is reached.

Bottom-up Synthesis. The second development methodology that our framework supports is a bottom-up development process through *stepwise composition*. Here we assume that actual implementations of some components already exist and that models are made that describe the behaviour of these components. The aim

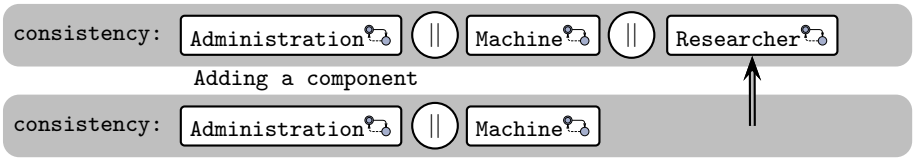

Fig. 5. One possible step in Bottom-up development: Checking consistency before and after adding an extra component.

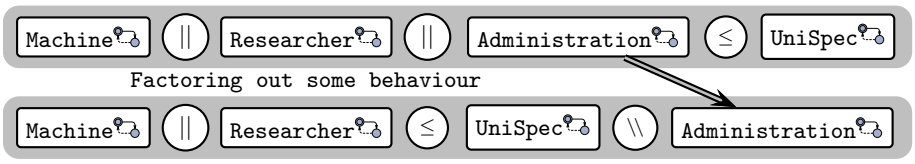

Fig. 6. One possible step in Stepwise modularisation of requirements: Factoring out the behaviour of the Administration.

of the bottom-up development in our setting is to verify that a complete system can be built from the preexisting components. Figure 5 shows one possible step in a bottom-up development process. Here a consistency check is performed on the parallel composition of two components after which another component is added and the consistency check is redone. The bottom up development methodology could easily be combined with refinement checking where the overall requirements are stepwise refined to see what the actual combination of components can guarantee in terms of behaviour and timing.

Stepwise Modularisation of Requirements. The third and more novel type of development methodology that our framework supports is the *Stepwise modularisation of requirements*. Here the idea is, like for the *top-down development* to start with a general specification of the requirements to the system and then using the *quotient* operator to factor out behaviour that is already implemented by existing components, so that one is left with a specification for the missing behaviour. This specification, which is synthesised by the tool, can now be further refined to provide the implementation of missing functionality in terms of new components. In that this process generalises stepwise refinement and bottom-up synthesis. Figure 6 shows how one component can be moved from one side of the refinement check to the other by factoring out the behaviour.

Another aspect of our framework that can be used orthogonally to the three described development ideas is conjunction. Conjunction allows to specify different aspects or requirements to a component and then compose these using logical conjunction, such that an implementation would have to individually satisfy each conjunct in order to satisfy the conjunction. Figure 7 shows an example of two specifications that each handle one aspect of the responsibilities of the Administration component. Figure 7a describes an alternation between the

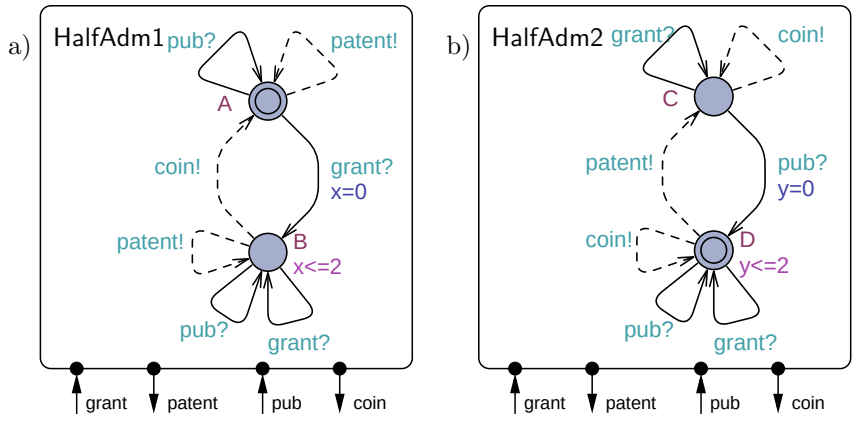

Fig. 7. Example of two conjuncts, each handling one aspect, that make up a different model of the Administration component.

coin! and grant? while Figure 7b describes the alternation between patent! and pub?. Together these form an alternative and slightly more loose specification of the administration. It is the case that both Administration and Administration2 refine the conjunction HalfAdm1∧HalfAdm2, while the opposite is not the case.

Finally the tool is able to verify TCTL* [1] properties on the specifications. This feature is made possible thanks to the modified underlying verification engine. This will be exemplified in section 5.

4 Combining Specification Automata

In this section, we discuss the three main operations defined on specification automata, namely: conjunction, composition, and quotient. All of these were used to support different design processes described in the previous section.

In the rest of the section, we will consider two specification automata $A_S = (Loc_1, q_0^1, Clk_1, E_1, Act^1, Inv_1)$ and $A_T = (Loc_2, q_0^2, Clk_2, E_2, Act^2, Inv_2)$. For technical reasons, we also assume that $Clk_1 \cap Clk_2 = \emptyset$.

4.1 Conjunction

Conjunction allows to test whether several specifications can be simultaneously met by the same component. In our framework, conjunction can only be defined if $Act_i^S = Act_i^T$ and $Act_o^S = Act_o^T$. The operation reduces to check whether the two specifications can progress in the same way. Formally the conjunction of A_S and A_T, denoted $A_S \wedge A_T$, is the TIOA $A = (Loc, q_0, Clk, E, Act^S, Inv)$ given by: $Loc = Loc_S \times Loc_T$, $q_0 = (q_0^S, q_0^T)$, $Clk = Clk_S \uplus Clk_T$, $Inv((q_S, q_T)) = Inv(q_S) \wedge Inv(q_T)$. The set of edges E is defined by the following rule:

– If $(q_S, a, \varphi_S, c_S, q'_S) \in E_S$ and $(q_T, a, \varphi_T, c_T, q'_T) \in E_T$ this gives rise to $((q_S, q_T), a, \varphi_S \wedge \varphi_T, c_S \cup c_T, (q'_S, q'_T)) \in E$.

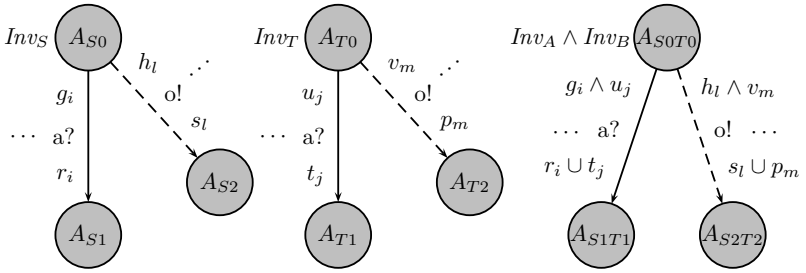

Fig. 8. Two states of TIOAs A_S and A_T are combined into one TIOA A with the conjunction operator for one step. This process is then iterated.

An example of how the conjunction of two specific states from two different TIOA with example input and output transitions will look is given in Figure 4.1.

Conjunction may introduce locally inconsistent states. For example, assume that A_S reaches a state from s where the only available action is the output a and A_T reaches a state t from where the only available action is the output b. Assume also that A_S and A_T cannot delay in s and t. In (s,t), the conjunction will not issue any output and will not be able to delay, which violates the *independent progress* property. As stated in Section 2, locally inconsistent states can be removed with the help of a pruning operation. In the rest of the paper, we assume that each conjunction is immediately followed by a pruning.

In [12], we have shown the following results:

- The set of implementations satisfying a conjunction is the intersection of the implementation sets of the operands: $[\![(A_S \wedge A_T)]\!]_{\text{mod}} = [\![A_S]\!]_{\text{mod}} \cap [\![A_T]\!]_{\text{mod}}$.
- The conjunction of A_S and A_T corresponds to the greatest lower bound of their implementations sets: if $A \leq A_S$ and $A \leq A_T$ we have that $A \leq A_S \wedge A_T$.
- The conjunction operation (also if combined with pruning) is associative and commutative, so among others: $[\![(A_S \wedge A_T) \wedge A_U]\!]_{\text{mod}} = [\![A_S \wedge (A_T \wedge A_U)]\!]_{\text{mod}}$.

4.2 Composition

We shall now define *structural composition*, also called *parallel composition*, between specifications. Roughly speaking, this operation computes the classical product between timed specifications [21], where components synchronise on common inputs/outputs. Two components are *composable* iff the intersection between their output alphabets is empty. Formally the *parallel composition* of A_S with A_T, denoted $A_S || A_T$, is the TIOA $A = (Loc, q_0, Clk, E, Act, Inv)$ given by: $Loc = Loc_S \times Loc_T$, $q_0 = (q_0^S, q_0^T)$, $Clk = Clk_S \uplus Clk_T$, $Inv((q_S, q_T)) = Inv(q_S) \wedge Inv(q_T)$ and the set of actions $Act = Act_i \uplus Act_o$ is given by $Act_i = Act_i^S \setminus Act_o^T \cup Act_i^T \setminus Act_o^S$ and $Act_o = Act_o^S \cup Act_o^T$. The set of edges E is defined by the following rules:

1. If $(q_S, a, \varphi_S, c_S, q'_S) \in E_S$ with $a \in Act_S \setminus Act_T$ then for each $q_T \in Loc_T$ this gives $((q_S, q_T), a, \varphi_S, c_S, (q'_S, q_T)) \in E$

2. If $(q_T, a, \varphi_T, c_T, q'_T) \in E_T$ with $a \in Act_T \setminus Act_S$ then for each $q_S \in Loc_S$ this gives $((q_S, q_T), a, \varphi_S, c_S, (q_S, q'_T)) \in E$
3. If $(q_S, a, \varphi_S, c_S, q'_S) \in E_S$ and $(q_T, a, \varphi_T, c_T, q'_T) \in E_T$ with $a \in Act_S \cap Act_T$ this gives rise to $((q_S, q_T), a, \varphi_S \wedge \varphi_T, c_S \cup c_T, (q'_S, q'_T)) \in E$.

The first rule represent all the cases where A_S makes an individual move, be it input or output, because a is not in the signature of A_T. Similarly the second rule handles all individual moves by the second component A_T. The third rule handles all synchronisations between the two components, no matter the combination of input and/or output. The rule is so simple because the type of the resulting transition is given by the sets Act_i and Act_o. The new output set, Act_o, is just a simple union of the outputs, while the input set, Act_i, is all the inputs that are not outputs of the other component.

Observe that if we compose two locally-consistent specifications using the above product rules, then the resulting product is also locally consistent. Moreover, unlike [16], our specifications are input-enabled, and there is no way to define an error state in which a component can issue an output that cannot be captured by the other component. The absence of "model-related" error states allows us to define more elaborated errors specified by the designer [12]. As an example, a temporal property written in some logic such as TCTL* can be interpreted over our specification, which when analysed by a model checker, will result in partitioning of the states into good ones (say satisfying the property) and bad ones (violating the property).

In contrast to conjunction, parallel composition is used to reason about external use of two (or more) components. We assume an independent implementation scenario, where the two composed components are implemented by independent designers. The designer of any of the environment components can only assume that the composed implementations will adhere to original specifications being composed. Consequently if an error occurs in parallel composition of the two specifications, the *environment* is the only entity that is possibly in a position to avoid it. Thus, each composition is followed by a pruning operation where all the states from which the environment has no strategy to avoid the set of bad states are removed.

In [12], we have shown the following important results regarding composition.

- Any implementation of composition can be realized by implementations of composed specifications: $[\![(A_S||A_T)]\!]_{\text{mod}} = [\![A_S]\!]_{\text{mod}}||[\![A_T]\!]_{\text{mod}}$.
- The composition operation (also if combined with a pruning) is associative and commutative, so among others: $[\![(A_S||A_T)||A_U]\!]_{\text{mod}} = [\![A_S||(A_T||A_U)]\!]_{\text{mod}}$.
- Refinement is a precongruence with respect to parallel composition; for any specifications A_S, A_T, and A_U such that $A_S \leq A_T$ and A_S composable with A_U, we have that A_T composable with A_U and $A_S||A_U \leq A_T||A_U$. Moreover if A_T compatible with A_U then A_S compatible with A_U.

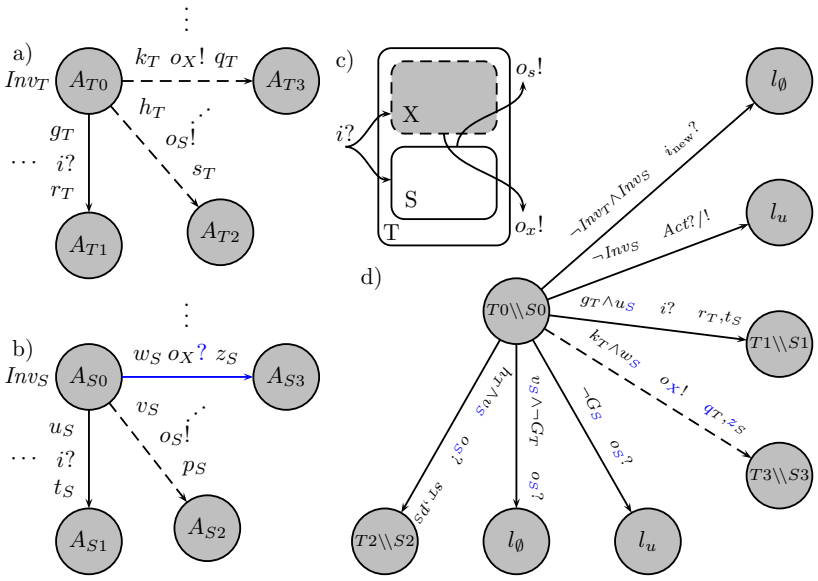

Fig. 9. Initial states of two example TIOA a) A_T, b) A_S, c) an overview of the communication flow and d) the initial state of the resulting quotient. Blue indicates later corrections to this figure, last updated 2020-08-25.

4.3 Quotient

An essential operator in a complete specification theory is the one of *quotienting*. It allows for factoring out behaviour from a larger component. If one has a large component specification A_T and a small one A_S then $A_T \backslash\backslash A_S$ is the specification of exactly those components that when composed with A_S refine A_T. In other words, $A_T \backslash\backslash A_S$ specifies the work that still needs to be done, given availability of an implementation of A_S, in order to provide an implementation of A_T. This is a non trivial operation which reduces to synthesis of a timed game. To the best of our knowledge, we are the first to compute the quotient in a timed setting.

We require that for the dividend A_T and the divisor A_S the following relations on action sets hold: $\Sigma_{\mathrm{i}}^S \subseteq \Sigma_{\mathrm{i}}^T$ and $\Sigma_{\mathrm{o}}^S \subseteq \Sigma_{\mathrm{o}}^T$. If these requirements on the input and output sets are met, then the quotient of A_T by A_S, which is denoted $A_T \backslash\backslash A_S$ is the TIOA given by: $Loc = Loc_T \times Loc_S \cup \{l_u, l_\emptyset\}$, $q_0 = (q_0^T, q_0^S)$, $Clk = Clk_T \uplus Clk_S \uplus \{x_{\mathrm{new}}\}$, $Inv((q_T, q_S)) = Inv(l_u) = true$ and $Inv(l_\emptyset) = \{x_{\mathrm{new}} \leq 0\}$. The two new states l_u and $l_\emptyset$ are respectively universal and inconsistent. The set of actions $Act = Act_i \uplus Act_o$ is given by $Act_i = Act_i^T \cup Act_o^S \cup \{i_{\mathrm{new}}\}$ and $Act_o = Act_o^T \setminus Act_o^S$. The set of edges E is defined by the following rules:

1. if $q_T \in Loc_T, q_S \in Loc_S$, $a \in Act$ then $((q_T, q_S), a, \neg Inv_S(q_S), \{x_{\mathrm{new}}\}, l_u) \in E$
2. if $q_T \in Loc_T, q_S \in Loc_S$ then $((q_T, q_S), i_{\mathrm{new}}, \neg Inv_T(q_T) \wedge Inv_S(q_S), \{x_{\mathrm{new}}\}, l_\emptyset) \in E$
3. if $(q_T, a, \varphi_T, c_T, q_T') \in E_T$ and $(q_S, a, \varphi_S, c_S, q_S') \in E_S$ this gives $((q_T, q_S), a, \varphi_T \wedge \varphi_S, c_T \cup c_S, (q_T', q_S')) \in E$
4. for each $(q_S, a, \varphi_S, c_S, q_S') \in E_S$ with $a \in Act_o^S$ this gives rise to $((q_T, q_S), a, \varphi_S \wedge \neg G_T, \{x_{\mathrm{new}}\}, l_\emptyset)$ where $G_T = \bigvee\{\varphi_T \mid (q_T, a, \varphi_T, c_T, q_T')\}$

5. if $(q_T, a, \varphi_T, c_T, q'_T) \in E_T$, $a \notin Act_S$ then $((q_T, q_S), a, \varphi_T, c_T, (q'_T, q_S)) \in E$
6. for each $(q_T, a, \varphi_T, c_T, q'_T) \in E_T$ with $a \in Act_o^S$ this gives rise to $((q_T, q_S), a, \neg G_S, \{\}, l_u)$ where $G_S = \bigvee\{\varphi_S \mid (q_S, a, \varphi_S, c_S, q'_S)\}$
7. for each $a \in Act_i$ this gives rise to $(l_\emptyset, a, x_{\text{new}} = 0, \{\}, l_\emptyset)$
8. for each $a \in Act$ this gives rise to $(l_u, a, \mathit{true}, \{\}, l_u)$

The quotients input set, Act_i, consists of the inputs to the outer component A_T and the outputs of the existing inner component A_S (See Figure 9c) and a new fresh input action i_{new}. The output set of the quotient, Act_o, is simply the output set of the outer component, A_T, minus the outputs handled by the existing inner component A_S. The resulting quotient has two new special states l_u and $l_\emptyset$. The first *universal* state, l_u, represents all the cases where the existing inner component A_s has violated the guarantees of the outer component A_T and thus there are no restrictions on the future behaviour of the quotient. The second *inconsistent* state, $l_\emptyset$, represents all the cases where the quotient by taking this action would itself violate the assumptions of the other components.

In the above definition we have eight rules. The first rule creates a new transition leading to the *universal* state with a guard that equals the original invariant of the existing inner specification. The second rule reflects the case where the invariant of A_T is not satisfied while the invariant of A_S is. The third rule handles all regular synchronisation where the guards of both components are satisfied. The fourth rule handles the case where the inner component generates an output at a time where it is not allowed by any of the matching guards in the outer component A_T. The fifth rule handles the cases where A_T takes an action which is not in the action set of A_S. The sixth rule represents all the cases where A_T takes an action which is not allowed by any of the matching guards in A_S thus leading to the universal state. Finally the seventh rule makes the $l_\emptyset$ state inconsistent and the eighth rule ensures that the universal state has all possible behaviour. Figure 9 illustrates one step in the computation of a quotient.
Like Conjunction, the quotient operation may produce (locally) inconsistent specifications. Hence, each quotient operation has to be followed by a pruning.

In [12], we have shown that the quotient operation produces the most liberal specification with respect to refinement. Formally we have the following theorem.

Theorem 6. *For any two specification automata A_S and A_T such that the quotient is defined, and for any implementation X over the same alphabet as $T \backslash\backslash S$ we have that $S||X$ is defined and $S||X \leq T$ iff $X \leq T \backslash\backslash S$.*

5 Tool Implementation

We begin with summarising the functionality of the tool, and proceed later to present a running example.

Our specification theory has been implemented as an extension of UPPAAL-TIGA [3], which is an engine for solving timed games. We have made two major modifications to the original engine. The first modification was to enrich the input language in order to allow for the description of specifications/implementations

and operations between them; the second one was to modify the timed-game algorithms in order to take the compositional reasoning methodology into account.

Modelling Language. The input syntax of UPPAAL-TIGA is identical to the one of UPPAAL [4, 26] – a tool for specifying, combining and verifying properties of timed automata. The user-friendly interface of UPPAAL is divided in two parts: 1) the *specification interface* where automata are specified in a graphical manner, and 2) the *query interface* where one can ask verification questions. The main difference between the input language of UPPAAL-TIGA and UPPAAL is that, due to the game interpretation, transitions are typed with control modalities. The specification interface of our tool is similar to the one of UPPAAL-TIGA, except that we decorate transitions with *input* and *output* modalities, which allows the user to specify timed I/O automata. Like timed automata, interfaces can communicate via broadcast channels, but global variables are not permitted. The query interface allows the user to (a) check whether a TIOA is a proper implementation or a specification, (b) to apply composition operations and (c) to check refinement relations. More details can be found at [36].

Each time we specify a TIOA, the tool automatically checks whether it is deterministic and input-enabled. In case of an implementation, the tool also checks whether it satisfies the output urgency and independent progress properties. Also, the tool automatically computes the set of states for which the specification is consistent. When the specification is combined with another one, this information is used in order to avoid involving bad states.

Timed Interface Operators. We have implemented the composition, conjunction, and refinement operators. Quotient is being implemented. These operators are available from the query interface. We now give details regarding the modifications we have made on the original version of UPPAAL-TIGA.

As we have seen, the operations of *conjunction*, *composition*, and *quotienting* may produce specifications with *bad* states. Such states need to be identified and pruned away. For doing so, we have adapted the game algorithm implemented in UPPAAL-TIGA. The main challenge, in terms of implementation, is that the original algorithms work on fixed input automata. In our case the automata are not known in advance since they result from the successive pruning operations.

The problem of checking whether A_S refines A_T reduces to the one of solving a timed game between two players on the graph-product of A_S and A_T [12]. The first player, or *attacker*, plays outputs on A_S and inputs on A_T, whereas the second player, or *defender*, plays inputs on A_S and outputs on A_T. One can show that Refinement does not hold if and only if the attacker can put the defender in a bad state. There are two kinds of bad states in this game: 1) the attacker may delay and violates invariants on A_T, which is, the defender cannot match a delay, and 2) the defender has to play a given action and cannot do so, i.e., a deadlock with respect to the game. In [8], we have proposed and implemented an efficient algorithm for solving such a game.

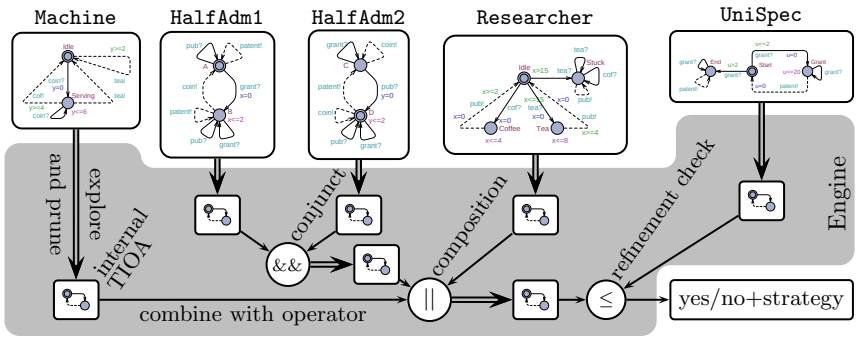

Fig. 10. Illustration of the steps performed in a concrete refinement check. The grey box represents the part carried out internally by the verification engine.

We illustrate the input language and the functionalities of the tool with the university example presented in Section 3. We fist consider the part of the example in which the administration is split in two parts (see Figure 7). We thus have four implementations (Machine, HalfAdm1, HalfAdm2, and Researcher) and one specification (UniSpec). All of these machines can easily be drawn in the specification interface. In the query interface, specifications and implementations are declared as follows: specification: UniSpec, and implementation: HalfAdm1. The tool automatically plays a safety-game followed by a pruning in order to remove locally inconsistent states. The tool also makes sure that the implementation and the specification satisfy Definitions 3 and 4. If this is not the case, or if the specification admit no implementation, then the tool stops.

We combine implementations with composition and conjunction operators as follows. The interface of the administration is the conjunction of two interfaces, one specifying when to output coins (after grants) and the other when to deliver a patent (after a publication). This is a better (and less restrictive approach) than to specify manually the combination of both. Then we compose the interfaces of the researcher, the university, and the machine. To check if this composition refines our original specification we check the following query

```
refinement: (Researcher || (HalfAdm1 && HalfAdm2) || Machine) <= UniSpec.
```

Figure 10 illustrates the different steps of the verification. The checker explores each component locally and prunes them from inconsistent states. The results of the exploration of the two "half-administrations" are conjuncted and pruned. Then the three automata are composed and pruned. The same is done for the specification and then the safety-game algorithm is used to check whether refinement holds. If at any step an automaton turns to be inconsistent then the check stops and the tool reports the error to the user. In this latter case, the user can invoke the simulator of Uppaal-tiga which will play the game until it breaks down. This information can be used to improve/change the design of the specification or of the implementation.

For the above example, it turns out that the refinement does not hold. The tool reveals that the UniSpec interface does not allow patents to be produced without a preceding grant. However, the composition allows researchers to publish with free tea, which is accepted by the conjunction of the two half administrations, which results in a patent. If we check instead

```
refinement: (Researcher || Administration || Machine) <= UniSpec.
```

with the administration of Figure 2b then the refinement holds as mentioned in Section 3 because this administration does not accept patents without grants first. However, specifying the administration manually exhibits a restricting behaviour that is not present in the cleaner conjunction of the two smaller specifications. Here the right correction would be to allow for free patents in UniSpec. The conclusion is that the user should not try to make the conjunction by hand and use conjunctions to specify more accurate specifications.

Finally, we illustrate the advantage of being capable to model check TCTL properties on TIOAs. We would like to avoid considering zeno behaviours. The idea is to combine our specification with an observer and then make sure that the observer visits infinitely often a state in which time advances. The latter can be specified with a TCTL property. The observer Obs has two states reset and advance and a witness clock w. The observer issues a non shared output from reset to advance if $w > 1$. Then it directly moves back to reset and resets the clock. The observer is then composed with the specification. There will be no synchronisation between the observer and the specification. Non zeno behaviours in the composition are those where Obs visits the state advance infinitely often. We use the following property that checks for refinement with an additional Büchi condition constraining the composition.

```
refinement: (Researcher || Administration2 || Machine || Obs
             : A[] A<> Obs.advance) <= UniSpec.
```

6 Related Work

In this section, we compare our results with other timed interface theories.

Input/Output automata model There have been several other attempts to propose an interface theory for timed systems (see [14, 16, 13, 7, 6, 10, 35, 17, 27] for some examples). Our model shall definitely be viewed as an extension of the timed input/output automaton model proposed by Lynch et al. [21]. The major differences are in the game-based treatment of interactions and the addition of quotient and conjunction operators.

Timed Interfaces by de Alfaro et al. In [16], de Alfaro et al. proposed *timed interfaces*, a timed extension of the interface model they introduced in [14]. Like for specification automata, the syntax of a timed interface is similar to the one of a *timed input/output automaton* [22] and the semantic is given by a timed

game. However, unlike specification automata, timed interfaces are not forced to be deterministic or input-enabled. The absence of input-enabledness allows for defined error states in the composition where one component can issue an output that cannot be captured by the other component. Two timed interfaces are said to be compatible if there exists an environment in which they can work together while avoiding such error states. This definition of compatibility allows to capture the timing between interfaces: "what are the temporal ordering constraints on communication events between components?". Unfortunately, the work in [16] is incomplete. Indeed there is no notion of implementation and refinement. Moreover, conjunction and quotient are not studied. Also, de Alfaro et al. did not consider more elaborated error states specified by the user with some timed temporal logic. Finally, the theory has only been implemented in a prototype tool called TICC [13], which does not handle continuous time. A main drawback of TICC is its textual input language that is far from modern graphical specification languages used by engineers.

Timed Modal Specifications In [23] Larsen proposes *modal automata*, which are deterministic automata equipped with transitions of the following two types: *may* and *must*. The components that implement such interfaces are simple labelled transition systems. Roughly, a must transition is available in every component that implements the modal specification, while a may transition need not be. Recently [7, 6] a timed extension of *modal* automata was proposed, which embeds all the operations presented in the present paper. However, modalities are orthogonal to inputs and outputs, and it is well-known [24] that, contrary to the game-semantic approach, they cannot be used to distinguish between the behaviours of the component and those of the environment. Aside from the orthogonality between input/output and may/must modalities. Our model does not allow to combine/compare automata that share common clock names, while in [7, 6] they restrict themselves to even-clock automata [2] for doing so. We are convinced that our theory directly extends to an event-clock automata version of TIOA with shared clocks. Finally, our work is implemented, while the work in [7, 6] is not implemented.

7 Conclusion

We have proposed a complete game-based specification theory for real time systems, in which we distinguish between a component and the environment in which it is used. To the best of our knowledge, our contribution is the first game-based approach to support both refinement, consistency checking, logical and structural composition, and quotient. Our results have been implemented in the Uppaal tool family [3].

In the future one could extend our model with global variables. This was already suggested by Berendsen and Vaandrager in [5], but only for structural composition and refinement and without the game-based semantic.

One could also investigate whether our approach can be used to perform scheduling of timed systems (see [13, 19, 17] for examples). For example, the

quotient operation could perhaps be used to synthesise a scheduler for such problems. It would also be of interest to add stochastic features to the model.

In [33, 25], we have proposed a model which takes advantages of both interface automata and modal specifications. One should follow a similar direction in the timed setting and combine our model with the one proposed in [7, 6].

Finally, our notion of error states is still primitive and in the future we plan to allow the users to define their own error states. This will be done with the help of some temporal logic, just like it was done for a refinement in [8].

References

1. R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.
2. R. Alur, L. Fix, and T. A. Henzinger. Event-clock automata: A determinizable class of timed automata. *Theoretical Computer Science*, 211:1–13, 1999.
3. G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime. Uppaal-tiga: Time for playing games! In *CAV*, volume 4590 of *LNCS*. Springer, 2007.
4. G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks. Uppaal 4.0. In *QEST*, pages 125–126. IEEE Computer Society, 2006.
5. J. Berendsen and F. W. Vaandrager. Compositional abstraction in real-time model checking. In *FORMATS*, volume 5215 of *LNCS*. Springer, 2008.
6. N. Bertrand, A. Legay, S. Pinchinat, and J.-B. Raclet. A compositional approach on modal specifications for timed systems. In *ICFEM*, LNCS. Springer, 2009.
7. N. Bertrand, S. Pinchinat, and J.-B. Raclet. Refinement and consistency of timed modal specifications. In *LATA*, volume 5457 of *LNCS*, Tarragona, Spain, 2009. Springer.
8. P. Bulychev, T. Chatain, A. David, and K. G. Larsen. Efficient on-the-fly algorithm for checking alternating timed simulation. In *FORMATS*, volume 5813 of *LNCS*, pages 73–87. Springer, 2009.
9. F. Cassez, A. David, E. Fleury, K. G. Larsen, and D. Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *CONCUR*, 2005.
10. K. Čerāns, J. C. Godskesen, and K. G. Larsen. Timed modal specification - theory and tools. In *Proceedings of the 5th International Conference on Computer Aided Verification (CAV'93)*, volume 697 of *LNCS*, pages 253–267. Springer, 1993.
11. A. Chakabarti, L. de Alfaro, T. A. Henzinger, and M. I. A. Stoelinga. Resource interfaces. In R. Alur and I. Lee, editors, *EMSOFT 03: 3rd Intl. Workshop on Embedded Software*, LNCS. Springer, 2003.
12. A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wąsowski. Timed I/O automata: a complete specification theory for real-time systems. In *HSCC*, 2010. Accepted.
13. L. de Alfaro and M. Faella. An accelerated algorithm for 3-color parity games with an application to timed games. In *CAV*, volume 4590 of *LNCS*. Springer, 2007.
14. L. de Alfaro and T. A. Henzinger. Interface automata. In *FSE*, pages 109–120, Vienna, Austria, Sept. 2001. ACM Press.
15. L. de Alfaro and T. A. Henzinger. Interface-based design. In *Marktoberdorf Summer School*. Kluwer Academic Publishers, 2004.
16. L. de Alfaro, T. A. Henzinger, and M. I. A. Stoelinga. Timed interfaces. In *EMSOFT*, volume 2491 of *LNCS*, pages 108–122. Springer, 2002.

17. Z. Deng and J. W. s. Liu. Scheduling real-time applications in an open environment. In *in Proceedings of the 18th IEEE Real-Time Systems Symposium, IEEE Computer*, pages 308–319. Society Press, 1997.
18. S. J. Garland and N. A. Lynch. The IOA language and toolset: Support for designing, analyzing, and building distributed systems. Technical report, Massachusetts Institute of Technology, Cambridge, MA, 1998.
19. T. A. Henzinger and S. Matic. An interface algebra for real-time components. In *IEEE Real Time Technology and Applications Symposium*, pages 253–266. IEEE Computer Society, 2006.
20. T. A. Henzinger and J. Sifakis. The embedded systems design challenge. In *FM*, volume 4085 of *LNCS*, pages 1–15. Springer, 2006.
21. D. K. Kaynar, N. A. Lynch, R. Segala, and F. W. Vaandrager. Timed i/o automata: A mathematical framework for modeling and analyzing real-time systems. In *RTSS*, pages 166–177. IEEE Computer Society, 2003.
22. D. K. Kaynar, N. A. Lynch, R. Segala, and F. W. Vaandrager. *The Theory of Timed I/O Automata*. Synthesis Lectures on Computer Science. Morgan & Claypool Publishers, 2009.
23. K. G. Larsen. Modal specifications. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 232–246. Springer, 1989.
24. K. G. Larsen, U. Nyman, and A. Wasowski. Modal I/O automata for interface and product line theories. In R. D. Nicola, editor, *ESOP*, volume 4421 of *LNCS*, pages 64–79. Springer, 2007.
25. K. G. Larsen, U. Nyman, and A. Wasowski. Modal i/o automata for interface and product line theories. In *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 64–79. Springer, 2007.
26. K. G. Larsen, B. Steffen, and C. Weise. Continuous modeling of real-time and hybrid systems: From concepts to tools. *STTT*, 1(1-2):64–85, 1997.
27. I. Lee, J. Y.-T. Leung, and S. H. Son. *Handbook of Real-Time and Embedded Systems*. Chapman, 2007.
28. N. Lynch. I/O automata: A model for discrete event systems. In *Annual Conference on Information Sciences and Systems*, pages 29–38, Princeton University, Princeton, N.J., 1988.
29. N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. Technical Report MIT/LCS/TM-373, The MIT Press, Nov. 1988.
30. O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems (an extended abstract). In *STACS*, pages 229–242, 1995.
31. R. Milner. *Communication and Concurrency*. Prentice Hall, 1988.
32. R. D. Nicola and R. Segala. A process algebraic view of input/output automata. *Theoretical Computer Science*, 138, 1995.
33. J.-B. Raclet, E. Badouel, A. Benveniste, B. Caillaud, A. Legay, and R. Passerone. Modal interfaces: unifying interface automata and modal specifications. In *EMSOFT*, pages 87–96. ACM, 2009.
34. E. W. Stark, R. Cleavland, and S. A. Smolka. A process-algebraic language for probabilistic I/O automata. In *CONCUR*, LNCS, pages 189–2003. Springer, 2003.
35. L. Thiele, E. Wandeler, and N. Stoimenov. Real-time interfaces for composing real-time systems. In *EMSOFT*, pages 34–43. ACM, 2006.
36. `http://www.cs.aau.dk/~adavid/tiga/tio.html`.
37. F. W. Vaandrager. On the relationship between process algebra and input/output automata. In *LICS*, pages 387–398, 1991.